第14章 继承与组合

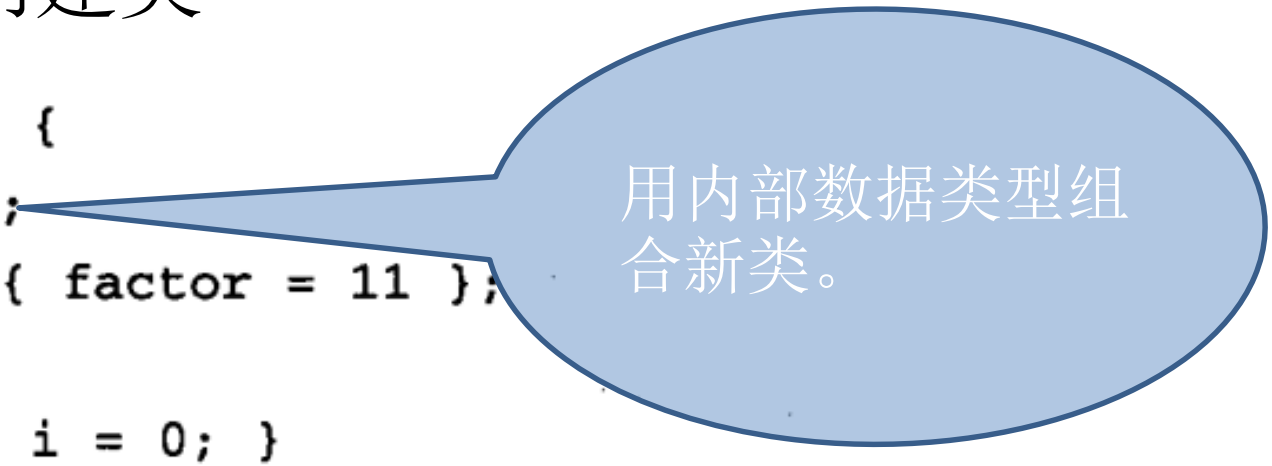
概述

- C ++最重要的特征之一是代码重用。应当能够做比拷贝代码更多的工作：在 C + +中，用类的方法解决，通过创建新类重用代码，就可以使用其他人已经创建并调试过的类。两种完成这件事的方法：
- 第一种方法很直接：简单地创建一个包含已存在的类对象的新类，因为这个新类是由已存在类的对象组合而成。所以称为组合（composition）。
- 第二种方法，创建一个新类作为一个已存在类的类型，不修改已存在的类，而采取这个已存在类的形式，并将代码加入其中，这个方法被称为继承（inheritance），其中大量的工作由编译器完成。继承是面向对象程序设计的基石

组合

- 用组合创建类

```
class X {
  int i;
  enum { factor = 11 };
public:
  X() { i = 0; }
  void set(int I) { i = I; }
  int read() const { return i; }
  int permute() { return i = i * factor; }
};
```



组合

- 组合一：将类型 X 的一个对象作为公共对象嵌入到一个新类内部，

```
class Y {
    int i;
  public:
    X x; // Embedded object
    Y() { i = 0; }
    void f(int I) { i = I; }
    int g() const { return i; }
  };

  main() {
    Y y;
    y.f(47);
    y.x.set(37); // Access the embedded object
  }
```

用用户定义类型组合新类。

访问嵌入对象（称为子对象）的成员函数只须再一次选择成员

组合

- 组合二：如果嵌入的对象是private的，可能更具一般性，这样，它们就变成了内部实现的一部分

```
class Y {
  int i;
  X x; // Embedded object
public:
  Y() { i = 0; }
  void f(int I) { i = I; x.set(I); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

main() {
  Y y;
  y.f(47);
  y.permute();
}
```

对于新类的public接口函数，包含对嵌入对象的使用，但不必模仿这个嵌入对象的接口。

继承

- 继承时，“这个新的类像原来的类”。我们规定，在代码中和原来一样给出该类的名字，但在这个类体的左括号前面，加一冒号和基类名（对于多重继承，要加多个基类名，它们之间用逗号分开）。这样做后，就自动地得到了基类中的所有数据成员和成员函数。

继承

```
class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;
  }
  void set(int I) {
    i = I;
    X::set(I); // Same-name function call
  }
};
```

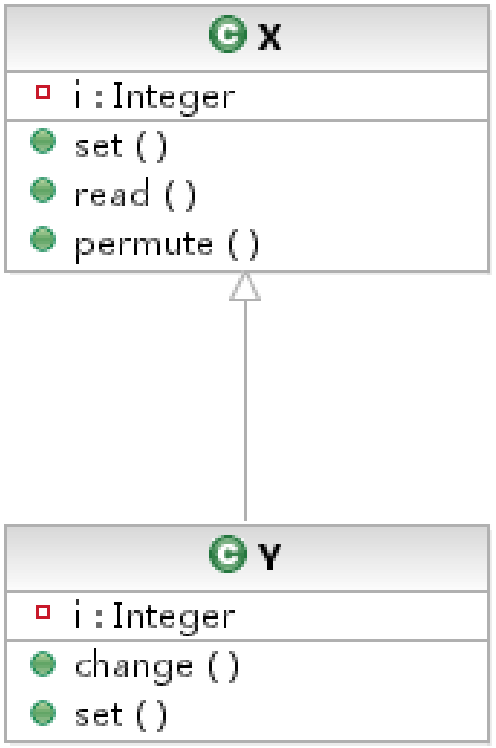
Y对X进行了继承，这意味着Y将包含X中所有的数据和方法

调用父类的函数

调用父类的函数

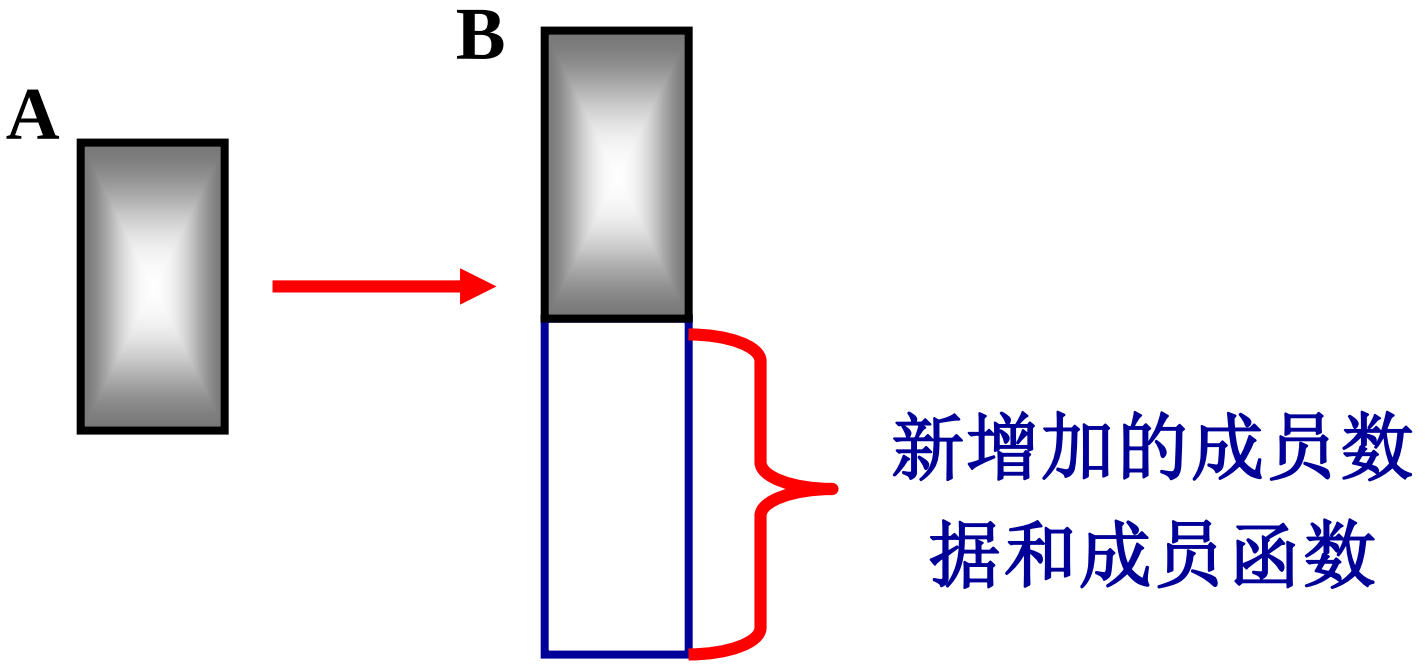
继承

```
main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
}
```



继承

类A派生类B：类A为基类，类B为派生类。



构造函数

- 问题回顾：构造函数的类型，构造函数的作用以及如何设计构造函数的参数？

- 如果一个类使用的组合或继承重用了其他类的代码，那么在构造这个类时，如何处理被重用的类的构造函数？

构造函数

- 构造函数的初始化表达式
  - 在继承的情况下，比如MyType继承了Bar，且Bar的构造函数只有一个int型参数，则构造函数如下：

```
MyType::MyType(int i) : Bar(i) { // ...
```

  - 在组合的情况下，成员的初始化也使用同样的语法：

```
MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...
```

构造函数

- 如何初始化基本数据类型，即那些没有构造函数的类型

```
class X {
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
  X x;
  int i(100);  // Applied to ordinary definition
  int* ip = new int(47);
} ///:~
```

构造函数

- 为了使语法一致，可以把内部类型看作这样一种类型：它只有一个带单个参数的构造函数。
- 这种伪构造函数由于具有良好和一致的编码风格，在创建内部类型的变量是也可以使用这种方法：

```
int i(100);
int* ip = new int(47);
```

组合与继承的联合

- 在下面的例子中，组合和继承被放在一起：

```
class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) {}
  ~C() {} // Calls ~A() and ~B()
  void f() const {  // Redefinition
    a.f();
    B::f();
  }
};

int main() {
  C c(47);
} ///:~
```

构造和析构的次序

- 一个对象在构造时，如果有父类和组合对象，其构造和析构的过程如下：
  - 构造时，先执行父类的构造过程，然后执行作为成员的组合对象的构造过程，最后执行自身的构造。
  - 析构时，与构造的过程相反。
  - 初始化列表中的初始化顺序不影响编译器对构造次序的决定。

构造和析构的次序

- 示例：C14:Order.cpp

```
#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);
```

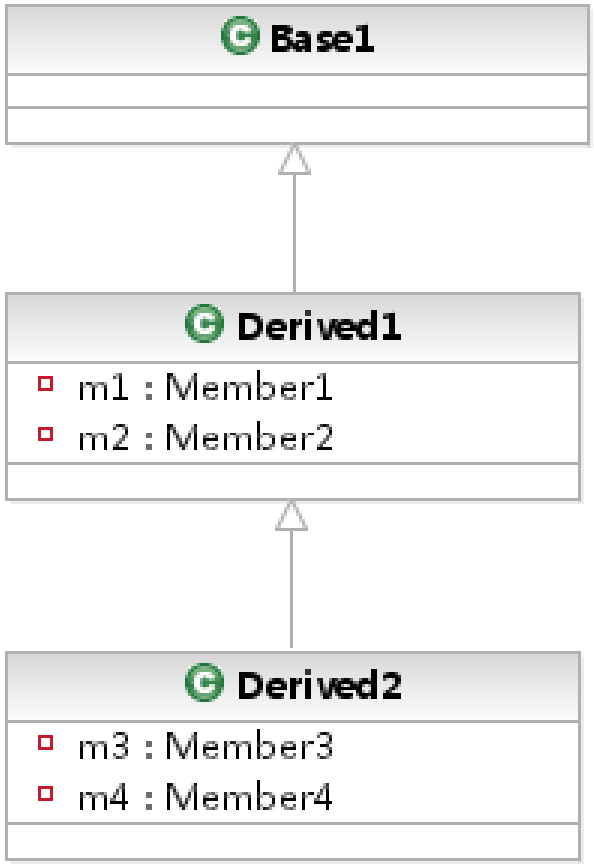
构造和析构的次序

```
class Derived1 : public Base1 {
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 constructor\n";
  }
  ~Derived1() {
    out << "Derived1 destructor\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 constructor\n";
  }
  ~Derived2() {
    out << "Derived2 destructor\n";
  }
};
```

```
int main() {
  Derived2 d2;
} ///:~
```

构造和析构的次序




```
int main() {
  Derived2 d2;
} ///:~
```

```
Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
Member1 destructor
Base1 destructor
```

继承：名字隐藏

- 回顾：函数重载
  - 一个类中，包含下面两个方法是合法的：
    - void print(char);
    - void print(float);
  - 在编译器内部，这两个函数被翻译成
    - _print_char和_print_float

继承：名字隐藏

- 继承中，如果子类中定义的一个函数，名称与父类相同，将发生什么？

```
class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};
```

```
class Derived1 : public Base {
public:
  void g() const {}
};
```

```
class Derived2 : public Base {
public:
  // Redefinition:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};
```

继承：名字隐藏

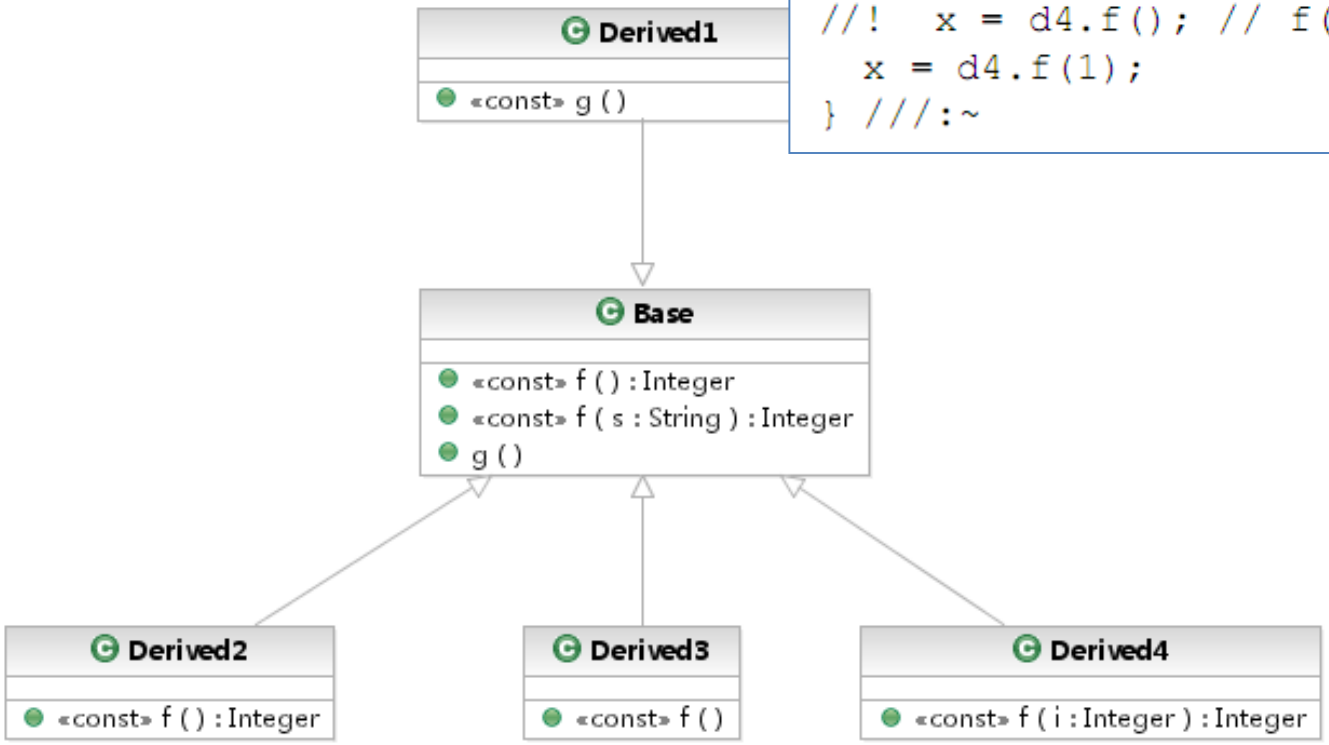
```
class Derived3 : public Base {
public:
  // Change return type:
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};
```

```
int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived3 d3;
//!  x = d3.f(); // return int version hidden
  Derived4 d4;
//!  x = d4.f(); // f() version hidden
  x = d4.f(1);
} ///:~
```

名字隐藏

```
int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived3 d3;
//!  x = d3.f(); // return int version hidden
  Derived4 d4;
//!  x = d4.f(); // f() version hidden
  x = d4.f(1);
} ///:~
```



继承：名字隐藏

- 结论：在子类中定义的函数将隐藏所有父类中有相同名称的函数。
- 如果希望访问那些定义在父类中被隐藏的方法，就必须通过转换，将子类作为父类来操作。

使用名字隐藏的继承示例：一个存放字符串的Stack

```
int main() {
  ifstream in("InheritStack.cpp");
  assure(in, "InheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) { // No cast!
    cout << *s << endl;
    delete s;
  }
} ///:~
```

使用名字隐藏的继承示例：一个存放字符串的Stack

```
class StringStack : public Stack {

public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
  ~StringStack() {
    string* top = pop();
    while(top) {
      delete top;
      top = pop();
    }
  }
};
```

push/peek/pop隐藏了Stack类中带void*类型或返回void*类型的函数

使用名字隐藏的继承示例：一个存放字符串的Stack

- 问题：
  - 这个类仅仅可以对string指针进行操作，如果想对某一其他类型的对象进行操作，只有实现一个新的Stack类。
  - 这个新的StringStack实际上是修改了原先Stack的接口。如果不能将StingStack当作Stack来使用，那我们为什么还需要继承呢？

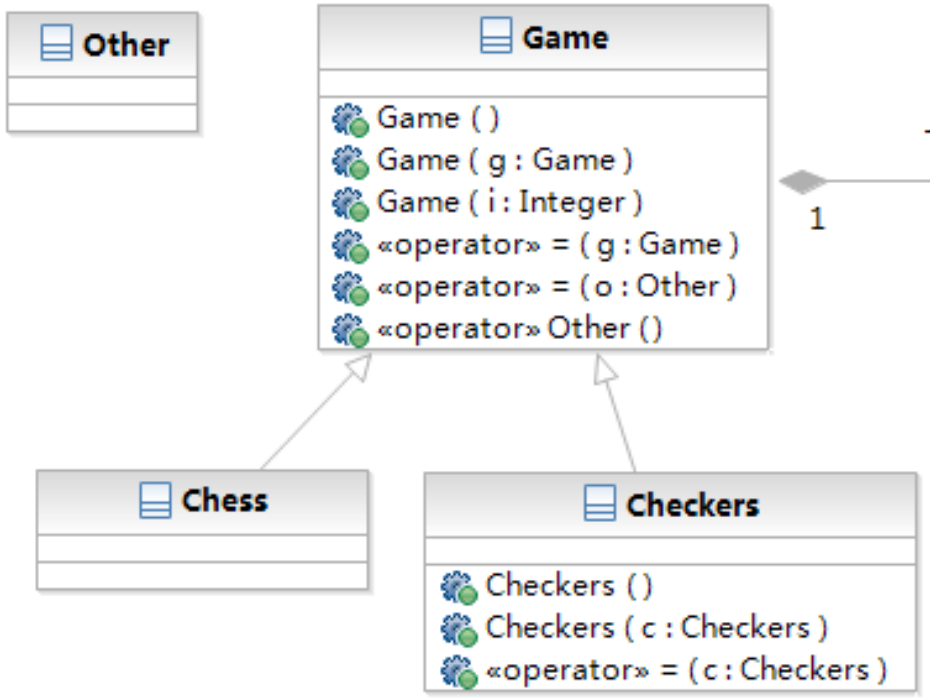
非自动继承的函数

- 继承的语义是，所有公共的函数都将复制到子类中（除非在子类中隐藏该函数）。
- 然而，有些函数，编译器知道父类和子类通常是不一样的，试图在子类中使用父类的函数往往是不好的习惯，或者会经常导致错误。这时候编译器会自动隐藏这些函数。

非自动继承的函数

- 构造函数
  - 父类和子类的构造函数通常不应该一样的，因为子类是父类的特化，应该包含更多的信息。
- operator=
- 示例：C14:SynthesizedFunctions.cpp

自动继承与非自动继承的函数




```
void f(Game::Other) {}
int main() {
  Chess d1;  // Default constructor
  Chess d2(d1); // Copy-constructor
//! Chess d3(1); // Error: no int constructor
  d1 = d2; // Operator= synthesized
  f(d1); // Type-conversion IS inherited
  Game game;
  Game::Other go;
  game = go;
//!  d1 = go; // Operator= not synthesized
           // for differing types
  Checkers c1, c2(c1);
  c1 = c2;
} ///:~
```

非自动继承的函数

- 在自定义拷贝构造函数和operator=时，如果需要调用父类对应的函数，必须在定义时显式地调用。
- 编译器不会像在生成缺省的拷贝构造函数和operator=时那样调用父类的对应函数。

```
class Checkers : public Game {
public:
Checkers(const Checkers& c) : Game(c) {
   cout << "Checkers(const Checkers& c)\n";
 }
 Checkers& operator=(const Checkers& c) {
   Game::operator=(c);
   cout << "Checkers::operator=()\n";
   return *this;
 }
};
```

继承与静态成员函数

- 静态成员函数在继承时的行为与非静态成员基本类似：
  - 它们均可被继承到派生类中
  - 如果我们定义了一个静态成员，所有在基类中的同名函数将被隐藏

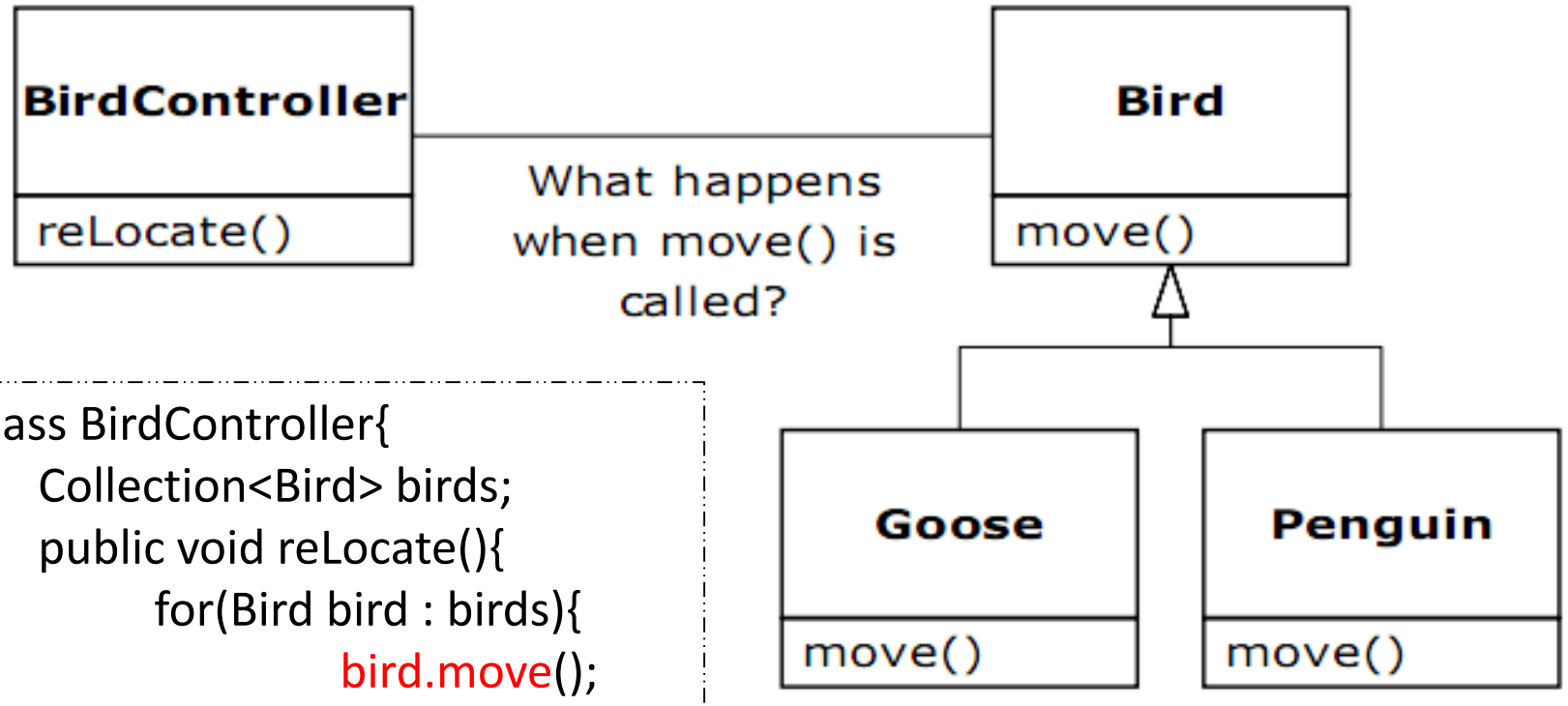
继承和组合的选择

- 如果一个对象只使用一个其他的对象，继承和组合看上去可以达到同样的效果：将一个对象放到另一个对象中，这时，可以按以下方式来选择继承或组合：
  - 如果只是希望使用新的对象的某些功能，而不是希望新的对象具有与另一对象拥有相同的接口，这时最好使用组合。
  - 如果希望新的对象具有与另一对象拥有相同的接口，则应该使用继承。

继承和组合的选择

- 优缺点
  - 组合优点：黑盒复用；不破坏封装性；扩展性较好；支持动态组合；
  - 组合缺点：整体类不能自动获得和局部类同样的接口； 创建整体类的对象时，需要创建所有局部类的对象；系统中对象过多；为了将不同的对象进行组合，必须仔细对接口进行定义
  - 继承优点：易于进行新的实现；已于修改或扩展被复用的实现
  - 继承缺点：白盒复用；破坏了封装性；父类的内部细节对子类可见；父类更改时，子类也必须随之更改

继承：相同的接口




```
class BirdController{
    Collection<Bird> birds;
    public void reLocate(){
        for(Bird bird : birds){
            bird.move();
        }
    }
}
```

组合示例

```
class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
  void inflate(int psi) const {}
};

class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};
```

```
class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
  Wheel wheel[4];
  Door left, right; // 2-door
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

组合示例

- 如果希望创建一个对象，不仅像 ifstream对象一样能够打开一个文件，而且含能保存文件名。使用组合方式的代码如下：

组合示例

```
class FName1 {
  ifstream file;
  string fileName;
  bool named;
public:
  FName1() : named(false) {}
  FName1(const string& fname)
    : fileName(fname), file(fname.c_str()) {
    assure(file, fileName);
    named = true;
  }
  string name() const { return fileName; }

  void close() { file.close(); }

  void name(const string& newName) {
    if(named) return; // Don't overwrite
    fileName = newName;
    named = true;
  }
  operator ifstream&() { return file; }
};
```

```
int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
  // Error: close() not a member:
//!  file.close();
} ///:~
```

使用继承

- 在上述示例中，程序员实际的用意是将新的类看作是ifstream。
- 在这种情况下，最好使用继承。

使用继承

```
class FName2 : public ifstream {
  string fileName;
  bool named;
public:
  FName2() : named(false) {}
  FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Don't overwrite
    fileName = newName;
    named = true;
  }
};
```

```
int main() {
  FName2 file("FName2.cpp");
  assure(file, "FName2.cpp");
  cout << "name: " << file.name() << endl;
  string s;
  getline(file, s); // These work too!
  file.seekg(-200, ios::end);
  file.close();
} ///:~
```

私有继承

- 在公有继承的情形下，父类对子类的可见性基本上与组合相同。
- 在私有继承的情形下，父类的所有成员对于该子类的用户不可见，该子类实际上不能被看作这个基类的实例（考虑接口）。
- 这种继承方式等价与在子类中声明一个私有的成员。
- 可以用这种方法隐藏基类的部分功能。

私有继承

基类成员属性	派生类	派生类外
公有	可以引用	不可引用
保护	可以引用	不可引用
私有	不可引用	不可引用

基类： public: (变为私有)在派生类中使用，类外不可使用

protected: (变为私有）在派生类中使用，类外不可使用

private: 不能在派生类中和类外使用

私有继承

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
  Pet::eat; // Name publicizes member
  Pet::sleep; // Both overloaded members exposed
};

int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Error: private member function
} ///:~
```

保护的成员

- 对于声明为保护的成员，被子类继承后，子类可见，子类的客户不可见。

```
class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};
```

保护继承

基类成员属性	派生类	派生类外
公有	可以引用	不可引用
保护	可以引用	不可引用
私有	不可引用	不可引用

基类：public: (变为保护)在派生类中使用，类外不可使用

protected: (变为私有）在派生类中使用，类外不可使用

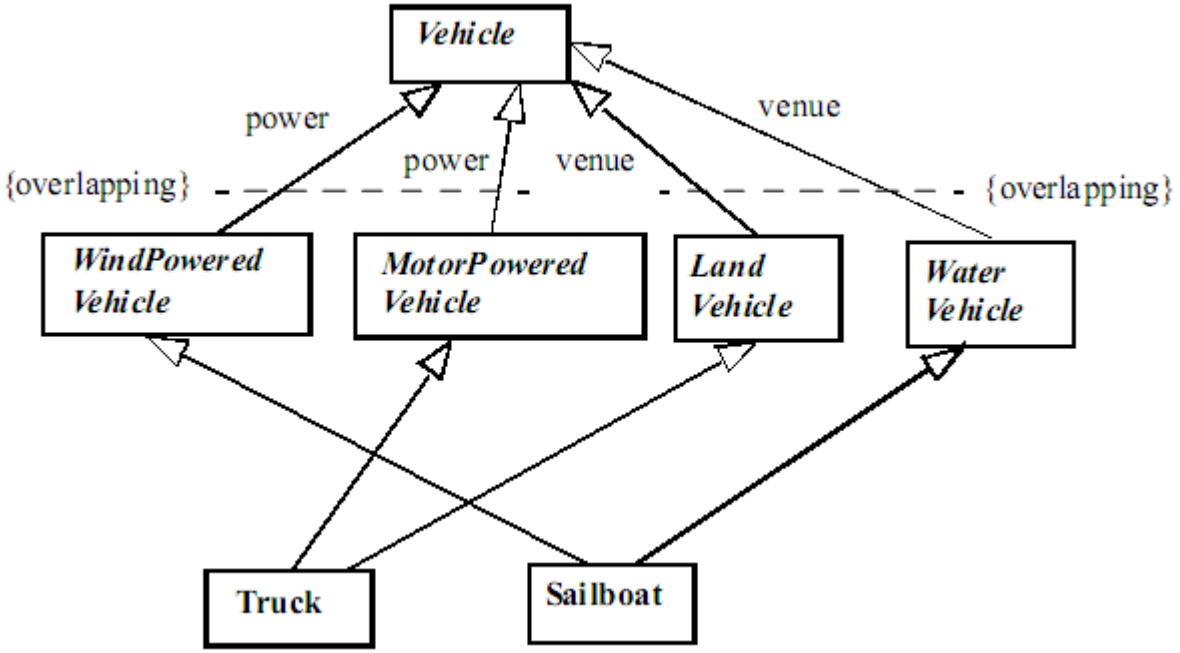
private: 不能在派生类中和类外使用

运算符的继承

- 除了赋值运算符外，其余的运算符可以自动地继承到派生类中。
- C12:byte.h

多重继承

- 滥用多重继承会使得程序变得过度复杂，而不是所期望的更加清晰。



渐增式开发

- 将软件开发的过程看作是一个使用简单的组件拼装为一个越来越大的系统的过程
- 每个组件都是经过仔细设计和测试，可以信赖的。
- 将软件系统看作是一个有机的，不断演化的生物，而不是将其看作是一个有着巨大的玻璃幕墙的摩天大厦。
- 组件越来越复杂的过程是一个逐渐向问题领域迈进的过程。

继承的目标：向上类型转换

- 继承的最重要的方面不是它为新类提供了成员函数，而是它与基类间建立起来的关系：新类属于原有的类型，能够把它当作原有的类型来使用。
- 将一个特殊的类型看作是一个一般的类型，称为向上类型转换。
- 向上类型转换示例：

向上类型转换

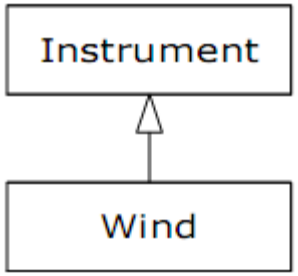
```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```



再论组合与继承

- 确定应当使用组合还是继承的方法之一是询问是否需要从新类向上类型转换。
- 在StringStack的例子中，由于StringStack需要重定义Stack的接口，并且在使用时将StringStack作为一个新的类来使用，不需要向上类型转换，所以更合适的方法可能是组合。

再论组合与继承：组合

```
class StringStack {
  Stack stack; // Embed instead of inherit
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
  }
  string* pop() {
    return (string*)stack.pop();
  }
};
```

再论组合与继承：继承

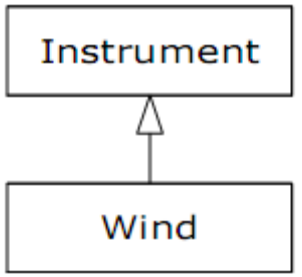
```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```



是Wind的play还是Instrument的Play被调用了？

小结

- 组合和继承有许多相似之处
  - 代码的复用
  - 初始化和清理的方式
- 继承的语法
  - 共有继承/私有继承
- 如何选择组合与聚合
  - 向上类型转换